Course: The XPath Language

Pierre Genevès
CNRS

University of Grenoble, 2012–2013

Why XPath?

Search, selection and extraction of information from XML documents are essential for any kind of XML processing.

→ XPath is the W3C standard language for expressing traversal and navigation in XML trees.

XPath Introduction

- A common syntax and semantics for many web languages
- A W3C recommendation (www.w3.org/TR/xpath)
- Compact syntax, not in XML, for use within XML attributes
- A language for expressing paths
- XPath operates on the logical (tree) structure of XML documents, not on their syntax

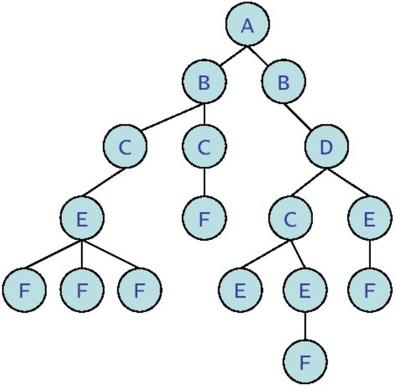
XPath Expressions

- XPath provides a powerful mechanism for navigating in XML trees: the *location path*
- A *location path* is a sequence of *location steps* separated by '/':

$$\underbrace{\texttt{child} :: \texttt{chapter}}_{\textit{location step}} / \underbrace{\overbrace{\texttt{descendant}}^{\textit{axis}} :: \overbrace{\texttt{section}}^{\textit{nodetest}}}_{\textit{location step}} / \underbrace{\texttt{child} :: \texttt{para}}_{\textit{location step}}$$

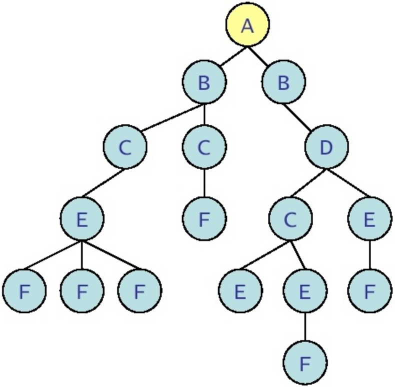
Evaluating a *location path*

- Starting from a context node, a *location path* returns a *node-set*
- Each node of this *node-set* becomes in turn the context node for evaluating the next *step*



Evaluating a *location path*

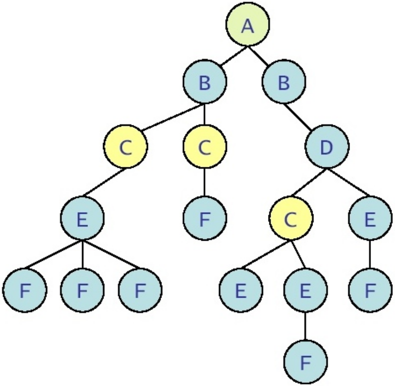
- Starting from a context node, a *location path* returns a *node-set*
- Each node of this *node-set* becomes in turn the context node for evaluating the next *step*



Context node

Evaluating a *location path*

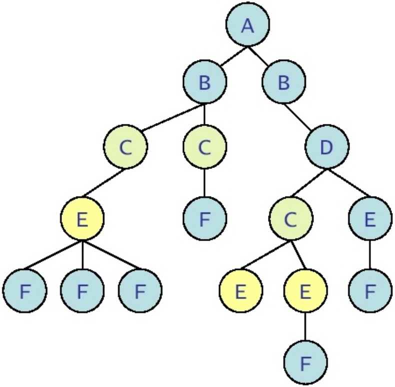
- Starting from a context node, a *location path* returns a *node-set*
- Each node of this *node-set* becomes in turn the context node for evaluating the next *step*



descendant::C

Evaluating a *location path*

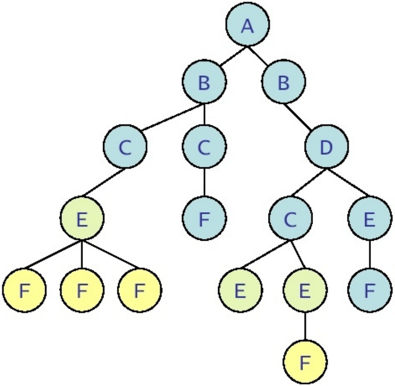
- Starting from a context node, a *location path* returns a *node-set*
- Each node of this *node-set* becomes in turn the context node for evaluating the next *step*



descendant::C/child::E

Evaluating a *location path*

- Starting from a context node, a *location path* returns a *node-set*
- Each node of this *node-set* becomes in turn the context node for evaluating the next *step*



`descendant::C/child::E/child::F`

Evaluation Context

- Every XPath expression is evaluated with respect to a *context* that includes:
  - the *context node*
  - 2 integers > 0 obtained from the evaluation of the last *step*:
    - *context size*: the number of nodes in the *node-set*
    - *context position*: the index of the context node in the *node-set*
  - a set of variable bindings (expressed in the host language)
- Navigation "propagates" the *context*: evaluation of a*step* yields a new *context state*
- Remark: a *location path* starting with '/' indicates that the initial *context* is set to the root of the document, such a *location path* is called "*absolute*"

Zoom on *location steps*

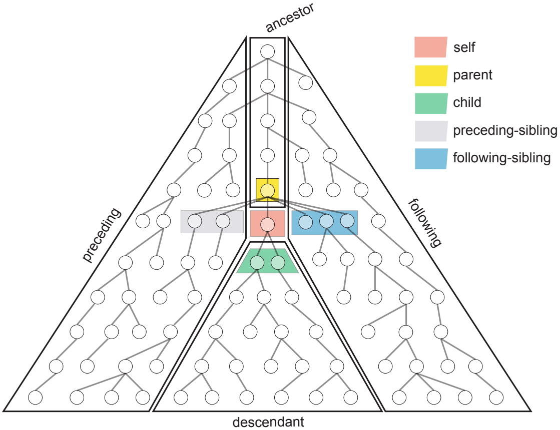
- A each navigation step, nodes can be filtered using *qualifiers*
- General syntax of a *location step*:

$$axis::nodetest[qualifier][qualifier]$$

- A *location step* is composed of 3 parts:
  1. an *axis*: specify the relation between the context node and returned nodes
  2. a *nodetest*: type and name of returned nodes
  3. optional *qualifiers* that further filter nodes
- Qualifiers are applied one after the other, once the selection is performed by the *axis* and *nodetest*
- A qualifier returns a *node-set* that is filtered by the next qualifier
- Example: `child::section[child::para][child::b]`

Axes

- Indicates where in the tree (with respect to the context node) selected nodes must be searched
- XPath defines 13 *axes* allowing navigation, including:



- 5 *axes* define a partition of tree nodes

Axes

- Each *axis* has a direction: forward or backward (w.r.t *document ordering*)
- Other axes:
  - `ancestor-or-self`, `descendant-or-self`
  - `attribute`: selects attributes of the context node (element)
  - `namespace`: selects namespace nodes of the context node

Nodetest

- The *nodetest* of a *location step* indicates which nodes must be chosen on the considered *axis*
- A *nodetest* filters nodes, e.g.:

Test	Semantics
`node()`	let any node pass
`text()`	preserve only text nodes
`comment()`	preserve only comment nodes
name	preserve only **elements/attributes** with tag "*name*"
`*`	preserve arbitrary **elements/attributes**

- Remarks:
  1. $\mathtt{\mathit{path}/child::*} \quad \subseteq \quad \mathtt{\mathit{path}/child::node()}$
  2. $\mathtt{\mathit{path}/attribute::node()} \quad \not\subseteq \quad \mathtt{\mathit{path}/child::node()}$

Qualifier

- A *qualifier* filters a *node-set* depending on the *axis* and returns a new*node-set*
- A *qualifier* is a boolean expression evaluated depending on the *context*:
  - context node
  - *context size*: number of nodes in the *node-set*
  - *context position*: index of the context node in the *node-set*, in the order of the document (or in reverse document order for *backward* axes)
- Each node of a *node-set* is kept only if the evaluation of the *qualifier* for this node returns *true*
- Examples:
  - `following-sibling::para[position()=last()]`
  - `child::para[position() mod 2 = 1]`

Value Comparisons

- *Qualifiers* may include comparisons:

$$\mathit{path}[\mathit{path}_1 \ \textbf{eq} \ \mathit{path}_2] \qquad \textbf{eq} \in \{=, \ != , \ <, \ >, \ <=, \ >=\}$$

- Existential semantics:

$$\textit{node-set}_1 \ \textbf{eq} \ \textit{node-set}_2$$
$$\text{iff}$$
$$\exists n_1 \in \textit{node-set}_1, \exists n_2 \in \textit{node-set}_2 \mid \texttt{string-value}(n_1) \ \textbf{eq} \ \texttt{string-value}(n_2)$$

Value Comparisons

- *Qualifiers* may include comparisons:

$$path[path_1\ \mathbf{eq}\ path_2] \qquad \mathbf{eq} \in \{=,\ !=,\ <,\ >,\ <=,\ >=\}$$

- Existential semantics:

$$node\text{-}set_1\ \mathbf{eq}\ node\text{-}set_2$$
$$\text{iff}$$
$$\exists n_1 \in node\text{-}set_1, \exists n_2 \in node\text{-}set_2 \mid \texttt{string-value}(n_1)\ \mathbf{eq}\ \texttt{string-value}(n_2)$$

- `string-value`(n): concatenation of all descendant text nodes in *document order*
- Example: `descendant::chapter[child::section="Conclusion"]`

→ all "`chapter`" nodes whose at least one "`section`" child has *string-value* "Conclusion".

Value Comparisons

- *Qualifiers* may include comparisons:

$$path[path_1 \ \mathbf{eq} \ path_2] \qquad \mathbf{eq} \in \{=,\ !=,\ <,\ >,\ <=,\ >=\}$$

- Existential semantics:

$$\textit{node-set}_1 \ \mathbf{eq} \ \textit{node-set}_2$$
$$\text{iff}$$
$$\exists n_1 \in \textit{node-set}_1, \exists n_2 \in \textit{node-set}_2 \mid \texttt{string-value}(n_1) \ \mathbf{eq} \ \texttt{string-value}(n_2)$$

- `string-value`(n): concatenation of all descendant text nodes in *document order*
- Example: `descendant::chapter[child::section="Conclusion"]`

→ all "`chapter`" nodes whose at least one "`section`" child has *string-value* "Conclusion".

- Comparisons may involve (implicit) type casting (ex: `a[b>7]`)

General XPath Expressions

- A general XPath *expression* is a *location path*, or a union of *location paths* separated by '|'
- *Qualifiers* may include boolean expressions:
  $path$[($path$ **eq** $path$) or ($qualifier$ and not($qualifier$))]
- An XPath expression may include *variables* (notation: $x)
  - variables are bound by the host language (*i.e.* they are constants ☺)
  - they are part of the evaluation context

Observation on Data Value Comparisons

- Assume variable $x is bound to a *node-set*
- What do you think of the following XPath expressions e_1 and e_2?

$$\underbrace{\texttt{\$x="foo"}}_{e_1} \qquad \underbrace{\texttt{not(\$x!="foo")}}_{e_2}$$

Observation on Data Value Comparisons

- Assume variable $x is bound to a *node-set*
- What do you think of the following XPath expressions e_1 and e_2?

$$\underbrace{\texttt{\$x="foo"}}_{e_1} \qquad \underbrace{\texttt{not(\$x!="foo")}}_{e_2}$$

- e_1 is different from e_2:

→ e_1 is true iff there exists a node in $x which has *string-value* foo;

→ e_2 is true iff all nodes in $x have string *string-value* foo.

Observation on Data Value Comparisons

- Assume variable \$x is bound to a *node-set*
- What do you think of the following XPath expressions e_1 and e_2?

$$\underbrace{\texttt{\$x="foo"}}_{e_1} \qquad \underbrace{\texttt{not(\$x!="foo")}}_{e_2}$$

- e_1 is different from e_2:

→ e_1 is true iff there exists a node in \$x which has *string-value* foo;

→ e_2 is true iff all nodes in \$x have string *string-value* foo.

- Owing to negation and comparison defined by existential quantification, we can formulate universal quantification...

Observation on Data Value Comparisons

- Assume variable $x is bound to a *node-set*
- What do you think of the following XPath expressions e_1 and e_2?

$$\underbrace{\texttt{\$x="foo"}}_{e_1} \qquad \underbrace{\texttt{not(\$x!="foo")}}_{e_2}$$

- e_1 is different from e_2:

→ e_1 is true iff there exists a node in $x which has *string-value* foo;

→ e_2 is true iff all nodes in $x have string *string-value* foo.

- Owing to negation and comparison defined by existential quantification, we can formulate universal quantification...
  - "`chapter`" nodes whose all children "`section`" are empty[1]?

[1] have an empty *string-value*

Observation on Data Value Comparisons

- Assume variable \$x is bound to a *node-set*
- What do you think of the following XPath expressions e_1 and e_2?

$$\underbrace{\texttt{\$x="foo"}}_{e_1} \qquad \underbrace{\texttt{not(\$x!="foo")}}_{e_2}$$

- e_1 is different from e_2:

→ e_1 is true iff there exists a node in \$x which has *string-value* foo;

→ e_2 is true iff all nodes in \$x have string *string-value* foo.

- Owing to negation and comparison defined by existential quantification, we can formulate universal quantification...
  - "`chapter`" nodes whose all children "`section`" are empty[1]?

  → `descendant::chapter[not(child::section!="")]`

[1] have an empty *string-value*

Basic Functions

- *Node-sets* are not the only types of XPath expressions: there are *boolean*, *numerical* and *string* expressions too
- Every XPath implementation must provide at least a list of basic functions called *Core Function Library* (c.f. appendix)
- Examples:
  - `last()`: a number, the *context size*
  - `position()`: a number, the *context position*
  - `count(`*node-set*`)`: number of nodes in the *node-set*
  - `concat(`*string*`, `*string*`, `*string**`)`: concatenate several strings
  - `contains(`*str1*`, `*str2*`)`: boolean, true if *str1* contains *str2*
  - ...
- Any XPath expression can be used within a *qualifier*, for instance:

```
descendant::recipe[count(descendant::ingredients)<5 and
contains(child::title, "cake")]
```

Abbreviated Syntax

- `child::` is the default axis, it can be omitted
- `@` is a shorthand for `attribute::`
- `//` is a shorthand for `/descendant-or-self::node()/`
- `.` is a shorthand for `self::node()`
- `..` is a shorthand for `parent::node()`
- `[4]` is a shorthand for `[position()=4]`

Example	Expanded Form
`book/section`	`child::book/child::section`
`p[@id="bla"]`	`child::p[attribute::id="bla"]`
`.//p`	`self::node()/descendant-or-self::node()/child::p`
`../title`	`parent::node()/child::title`
`p[3]`	`child::p[position()=3]`

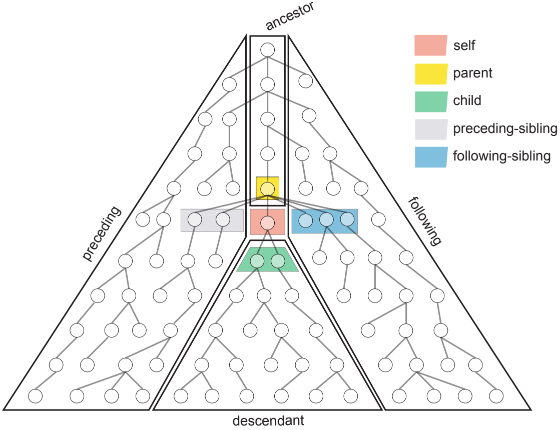
Question...

What do you think of the following XPath expressions e_1 et e_2?

$$\underbrace{\texttt{self::title}}_{e_1} \qquad \underbrace{\texttt{parent::node()/child::title}}_{e_2}$$

Question...

Can we rewrite the XPath expression `following::p` without the axis `following`?



XPath: A Core Component for XML Technologies

- XPath is used in:
  - XSLT: selection of document parts to be transformed
  - XQuery: XPath is the (main) subset of the query language
  - XPointer: identification of XML fragments
  - XLink: definition of hypertext links
  - XML Schema: expressing the tree region in which unicity is guaranteed
  - XForms: expressing dependencies (data bindings)
  - ...
- Often, it is even the **essential** component

XPath and Static Analysis (1/2)

- Many different ways to express navigation to the same nodes
- Two XPath expressions might share the same semantics[2] even if they differ syntactically (and operationally!)

  ```
  child::a[child::b]/following-sibling::c
  ```

  ```
  child::c[preceding-sibling::a[child::b]]
  ```

- Determining query equivalence is crucial (e.g. optimization)

[2] The semantics of an XPath expression is to be understood as the final set of nodes resulting from the evaluation of the expression.

XPath and Static Analysis (2/2)

- What about the following expressions?

  ```
  descendant::d[parent::b]/following-sibling::a
  ```

  ```
  ancestor-or-self::*/descendant-or-self::b/a[preceding-sibling::d]
  ```

- Question for next time(s): how would you write a program that checks whether two XPath expressions are equivalent (i.e. return the same set of nodes when applied from the same context in any tree)?

XPath *Core Function Library*

Functions over *node-sets*

- `last()`: a number, the *context size*
- `position()`: a number, the *context position*
- `count(`*`node-set`*`)`: number of nodes in the *node-set*
- `id(`*`object`*`)`: selects elements by their unique ID
- `local-name(`*`node-set`*`)`: returns the local part of the expanded-name of the node in the argument *node-set* that is first in document order.
- `namespace-uri(`*`node-set`*`)`: returns the namespace URI of the expanded-name of the node in the argument *node-set* that is first in document order
- `name(`*`node-set`*`)`: returns a string containing the whole name of the node in the argument *node-set* that is first in document order

String Functions

- `string(`*object*`)`: convert *object* to a string
- `concat(`*string*`, `*string*`, `*string**`)`: concatenate several strings
- `start-with(`*string1*`, `*string2*`)` : boolean, true if *string1* starts with *string2*
- `contains(`*str1*`, `*str2*`)` : boolean, true if *str1* contains *str2*
- `substring-before(`*string1*`, `*string2*`)`: the substring of *string1* before the first occurrence of *string2*
- `substring-after(`*string1*`, `*string2*`)`: the substring of *string1* after the first occurence of *string2*
- `substring(`*string*`, `*number1*`, `*number2*`)`: the substring of *string* that starts at position *number1* and whose length is *number2*
- `string-length(`*string*`)`: number of characters in *string*
- `normalize-space(`*string*`)`: remove beginning, ending and double spaces
- `translate(`*s1*`, `*s2*`, `*s3*`)`: replace in *s1* each char of *s2* by the char of same position in *s3*
  example : `translate("bar","abc","ABC")` returns `BAr`

Boolean Functions

- `boolean(`*`object`*`)`: convert *object* into boolean, returns true if non zero number, non empty *node-set*, string with non zero length
- `not(`*`boolean`*`)`: negation of *boolean*
- `true()`
- `false()`
- `lang(`*`string`*`)`: the language (attribute `xml:lang`) of context node is the same or a sublanguage of *string*

Arithmetic Functions

- `number(`*object*`)`: convert *object* into a number
- `sum(`*node-set*`)`: sum of the (type casted) number representation of each node in the *node-set*
- `floor(`*number*`)`: greatest integer less or equal to *number*
- `ceiling(`*number*`)`: smallest integer greater than or equal to *number*
- `round(`*number*`)`: the closest integer of *number*

Operator Precedence

1. `<=`, `<`, `>=`, `>`
2. `=`, `!=`
3. `and`
4. `or`